Heuristic Optimization

Implementation exercises1

Maxime Langlet

April 2021

Introduction

During these implementation exercises we had the chance to implement iterative improvement algorithms for the PFSP. We applied these algorithms to all instances of 50 and 100 jobs. We will then compare the different methods with a statistical analysis.

1 Exercise 1

1.1 Code Implementation and additions

Since initially, we are given some code with already implemented the Uninformed Random Picking as starting solution, a seed on the random picking was added since it was required to reduce the variance for the experiments as the algorithms need to start from the same initial solution. Furthermore, a couple of getters was added in the *pfspinstance* class since for the simplified RZ heuristic it is required to compute the Job matrix, so these getters return the job matrix and the priorities. Lastly, a python file to run the code on all instances at once was created, giving all the outputs in a single output file. To run the code, this simple command will do : **python src/make.py**, that is if all folders are correctly placed. An example of running the code on a single instance is the following : **./flowshopWCT ../instances/50_20_01 first transpose random**.

Simplified RZ Heuristics

The implementation of the simplified RZ heuristic starts by taking the job matrix and summing each lines and dividing by their respective priorities, outputting the result in a list(a line in the instances files represent a job contrary to the examples in the slides). In parallel, another list with each indexes of jobs is created. So the list constituted by the weighted sum of processing times of jobs will be sorted in ascending order, thus the list of indices will follow the same ordering.

We then have our initial solution, but it is not finished. We must also reorder the jobs to have a minimal weighted sum of completion time. Thus, starting with the first two jobs of our preceding list, we need to order them by the order minimizing this sum, then add a job and repeat that until the list is complete but keeping the order of the preceding iteration. For example, if $J_1 J_3$ was the preceding iteration, the next one we'll need to find the order with minimal WCT between $J_2 J_1 J_3$, $J_1 J_2 J_3$ and $J_1 J_3 J_2$.

Iterative improvements

All pivoting algorithms follow this pseudo code, where π' is a neighbor of π:

```
Algorithm 1: Iterative Improvement
Result: Improved solution
$\pi$ := GenerateInitialSolution();
while $\pi$ is not a local optimum do
    choose a neighbour $\pi\prime \in N(\pi)$ such that $F(\pi\prime) < F(\pi)$;
    $\pi := \pi\prime$;
end
```

Transpose : This rule is the most straight forward one since, from an initial solution, we just need to exchange two neighboring nodes. A simple for loop is capable of achieving that for us, all the nodes and his neighbor will be transposed. After each exchange, the WCT is computed and kept if it is better than the previous best. In the case of the first improvement, a break will stop this loop to and evaluate the while condition, determining if we must continue or if we stop. In the case of the best improvement, this break won't happen and the for loop will finish. To transpose two nodes the **iter_swap** function from the *algorithm* library was used.

Insert : To implement this rule, a double for loop is required since, for each nodes, we'll need to insert them at each other positions. To insert a node at a given position, we'll use the **rotate** function also from the *algorithm* library. Similarly as before, if the first improvement is selected, the for loop is ended as soon as an improvement, else we let the double loop finish.

Exchange : The exact same algorithm is used as *Insert* but only changing the function on the vectors exchanging rather than inserting now. We'll reuse the **iter_swap** function, but now if the indices from our double loop are equal, we'll skip the exchange. The rest is the same so will not be explained again.

1.2 Results

Pivoting rule

Here bellow shows the tables presenting the average percentage deviation and computation time for each of the iterative improvement algorithms comparing the pivoting rule. Note that the second table contains two time per cell, those two times represent the average computation time per number jobs, so 100 and 50 jobs respectively, the bigger number representing 100 jobs naturally.

Rule	First	Best
Transpose	35.880	36.812
Exchange	3.419	4.316
Insert	2.486	3.302

Table 1: Average percentage deviation with random initial solution

Rule	First	Best
Transpose	0.733/0.077	0.709/0.086
Exchange	4.329/0.355	40.470/2.281
Insert	5.851/0.531	45.444/2.707

Table 2: Average computing time with random initial solution in seconds

Remark that a rule of inverse proportionality between the performance in the average percentage deviation and the average computation time. Note that the transpose algorithm is the worst performing one but with a huge lead in computation time. The exchange and insert algorithms are quite close both with a lead in performance for the insert but costing a little bit more time. Another remark is that, given these numbers we don't see any particular gain using the best improvement rule. In our case, we even lose some performance using it. Therefore, using first improvement on our instances will not only reduce considerably the computation time, but also slightly increase the average derivation percentage. Therefore, in this testing configuration, first improvement is the better performing pivoting rule.

Initial solution

Here bellow shows the tables presenting the average percentage deviation and computation time for each of the iterative improvement algorithms comparing the generation of the initial solution. To get these results, the best improvement rule was used to see if there's a particular difference in computation time.

Algorithm	Random	SRZ
Transpose	36.812	4.162
Exchange	4.316	3.149
Insert	3.302	2.244

Table 3: Average percentage deviation with best iterative improvement

Algorithm	Random	SRZ
Transpose	0.709/0.086	1.245/0.092
Exchange	40.470/2.281	9.778/0.599
Insert	45.444/2.707	14.381/0.865

Table 4: Average computing time ins seconds with best iterative improvement

The global takeaways from before are also applying here, therefore computation time increases if we want a better average relative percentage deviation. The iterative improvement benefiting the most from the *SRZ* method is the transpose method. We almost have a 9 times better performing configuration than before, for slightly more computation time. We also see improvement for the two other iterative improvement methods in relative percentage deviation **and** in computation time. So the *SRZ* method seem better overall.

Further testing

Giving our preceding results, it is interesting to compare the better performing configurations between both categories, that is computation time and average relative percentage deviation. Those configurations both contain first improvement as pivoting rules and *SRZ* as initial solution. We have this table summarizing the observations :

Algorithm	Computation time (s)	ARPD
first+transpose+SRZ	0.670/0.091	4.159
first+insert+SRZ	3.974/0.367	2.192

Table 5: Comparison of the two best performing configurations

Both can be useful in their respective branches. If computing time is not an issue, we might use the insert method, giving us a better deviation. If on the contrary we have a time constraint, the transpose method might be better.

Until now, it was not discussed if there was a significant difference between the solution quality generated by the different algorithms. To that end we will evaluate the p-value of the Wilcoxon signed-rank test and the Paired t-test. Will apply these test on the average percentage deviation of each rules, with random initial solution and first improvement. Here's the table resuming the results:

Algorithm	Wilcoxon	T-test
Tranpose+Insert	1.671e-11	6.782e-49
Transpose+Exchange	1.671e-11	1.039e-47
Insert+Exchange	8.930e-07	9.280e-08

Table 6: Statistical tests p-value

Given these results, we can safely reject the null hypothesis and conclude that the two sets are statistically different. Other results from other configurations will not be represented because the same conclusion arise.

2 Exercise 1.2

2.1 Coding Implementation

For this second exercise, it required to implement a variable neighborhood descent (VND) algorithm, where two different neighborhood ordering will be tested. The pseudo-code for this VND algorithm is the following :

Algorithm 2: Variable neighborhood descent (VND)

```
Result: VND Solution
k neighborhoods $N_1, ..., N_k$;
$\pi = GenerateInitialSolution()$;
i=1;
while $i<k$ do
    choose the first improving neighbor $\pi\prime \in N_i(\pi)$;
    if $\nexists \pi\prime$ then
        $i$ += 1;
    else
        $\pi = \pi\prime$;
        $i = 1$;
    end
end
```

For this algorithm, the exact same iterative improvement methods are used from the preceding exercise in choosing an improving neighbor, so these won't be detailed again. Note that, as soon as an improvement is found, the first iterative improvement method will be called next (in our case, the first being Transpose). To that end, a jump label is used, returning to the beginning of the algorithm. So no loops were used in our case, here's a pseudo code of what the algorithm looks like :

Algorithm 3: Alternative VND implementation

```
Result: VND Solution
k neighborhoods $N_1, ..., N_k$;
$\pi = GenerateInitialSolution()$;
label;
$\pi\prime = Transpose(\pi)$;
if $F(\pi\prime) < F(\pi)$ then
    $\pi = \pi\prime$;
    goto label;
end
$\pi\prime = Exchange(\pi)$;
if $F(\pi\prime) < F(\pi)$ then
    $\pi = \pi\prime$;
    goto label;
end
$\pi\prime = Insert(\pi)$;
if $F(\pi\prime) < F(\pi)$ then
    $\pi = \pi\prime$;
    goto label;
end
```

Of course, when testing the Transpose, Insert and Exchange order, the two functions are inverted in the pseudo code above, and F is the cost function determining the WCT.

2.2 Results

In this section, a comparison between the two initialization methods are done. Here bellow shows the measurements :

Algorithm	Random	SRZ
VND_tei	2.894	2.202
VND_tie	2.962	2.349

Table 7: Average percentage deviation for different initial solution

Algorithm	Random	SRZ
VND_tei	49.223/3.039	19.853/1.284
VND_tie	47.746/2.717	13.367/0.851

Table 8: Average computation time for different initial solution

As we can notice from these tables, not only the *SRZ* method yields better percentages but also computes faster. To evaluate the percentage improvement, we'll use the following formula

$$\frac{ARPD\_VND - ARPD\_ii}{ARPD\_ii}$$

, where $ARPD\_VND/ii$ is the average over all ARPD for respecfully the VND algorithm and Iterative improvements algorithms. The VND algorithm yields on average a 0.8% uplift in ARPD performance compared to a single Iterative improvement with random initial solution and a 14% uplift for *SRZ* initial method. Lastly, the performances indicates that the better configuration is the VND algorithm with order transpose, exchange and then insert, but at the cost of computation time. If computation time has more weight, a solution could be to use the other ordering.

Now applying the same statistical tests from before we have these results:

Algorithm	Wilcoxon	T-test
VND+random	0.127	0.085
VND+SRZ	0.794	0.634

Table 9: Statistical tests p-value

Now a different behavior is shown compared to before. We can't with absolute certainty reject the null hypothesis. So there is a possibility that the two sets created by the configurations of the VND algorithm with random initialization are statistically correlated. With the *SRZ* method, it is even probable that their are correlated given these results.